Algorithms and Data Structures 2
CS 1501

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez’s and Dr. Farnan’s CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 8 and Lab 7: Tuesday 3/21 @ 11:59 pm
  - Homework 9: this Friday @ 11:59 pm
  - Assignment 3: Friday 3/31 @ 11:59 pm
    - Support video and slides will be on Canvas
    - Debugging tips

Previous lecture

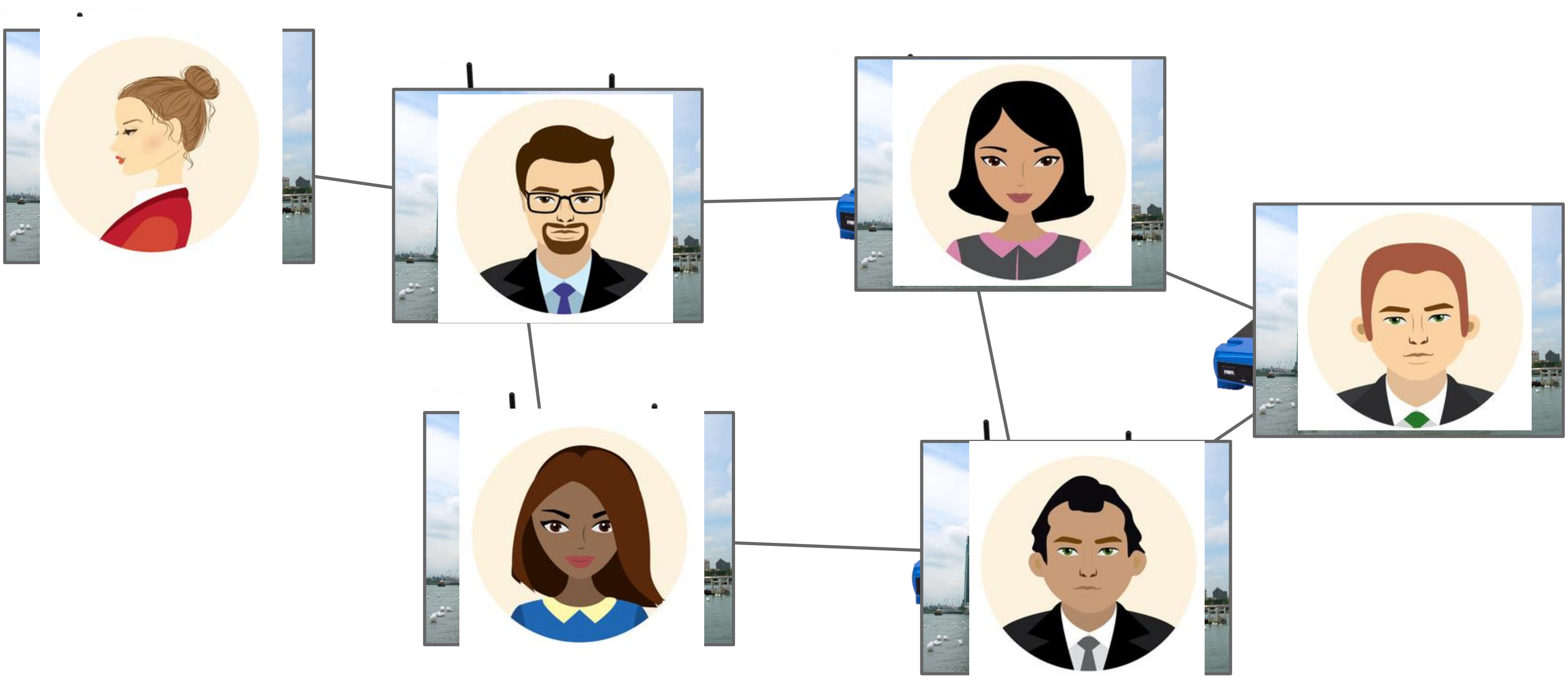
- Burrows-Wheeler Compression Algorithm
- ADT Graph
  - definitions
  - representation using array of linked lists

This Lecture

- ADT Graph
  - definitions
  - representations
    - two-arrays
    - adjacency matrix
    - adjacency lists
  - traversals
    - BFS
      - shortest paths based on number of edges
      - connected components
    - DFS
      - finding articulation points of a graph

Why?

- Can be used to model many different scenarios



Some definitions

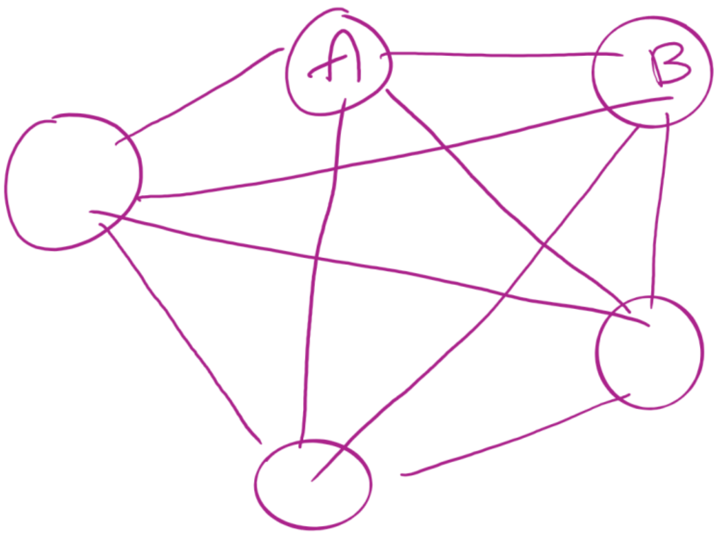
- Undirected graph
  - Edges are unordered pairs: (A, B) == (B, A)
- Directed graph
  - Edges are ordered pairs: (A, B) != (B, A)
- Adjacent vertices, or neighbors
  - Vertices connected by an edge

Graph sizes

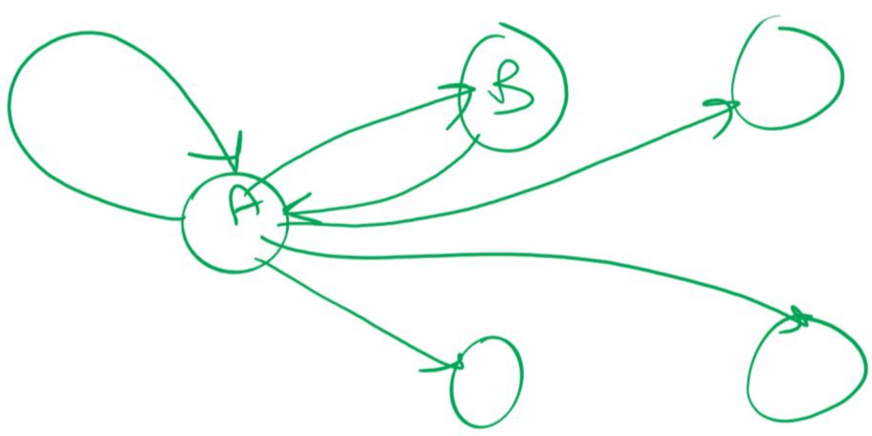
- Let v = |V|, and e = |E|
- Given v, what are the minimum/maximum sizes of e?
  - Minimum value of e?
    - Definition doesn't necessitate that there are any edges...
    - So, 0
  - Maximum of e?
    - Depends...
      - Are self edges allowed?
      - Directed graph or undirected graph?
    - In this class, we'll assume directed graphs have self edges while undirected graphs do not

Maximum value of e (MAX)

- Undirected graph
  - no self edges
  - v*(v-1)?
  - But, A->B is the same edge as B-> A
  - Are we counting each twice?
  - v*(v-1)/2

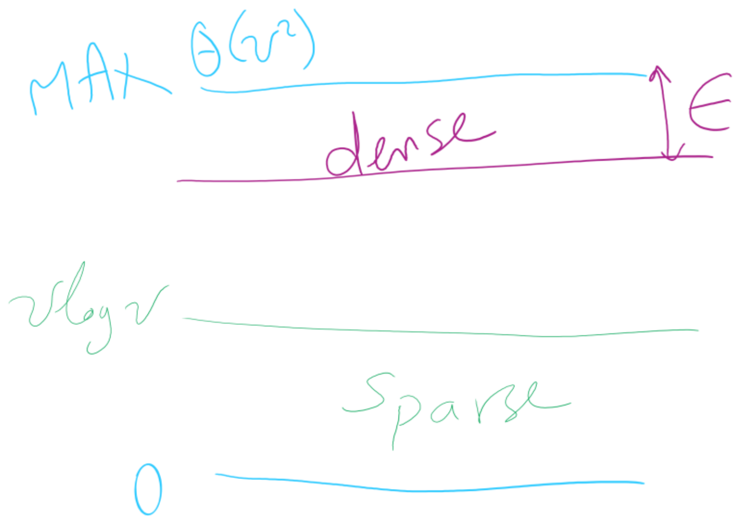


- Directed graph
  - self edges allowed
  - v*v?
  - A -> B is a different edge than B -> A
  - v^2

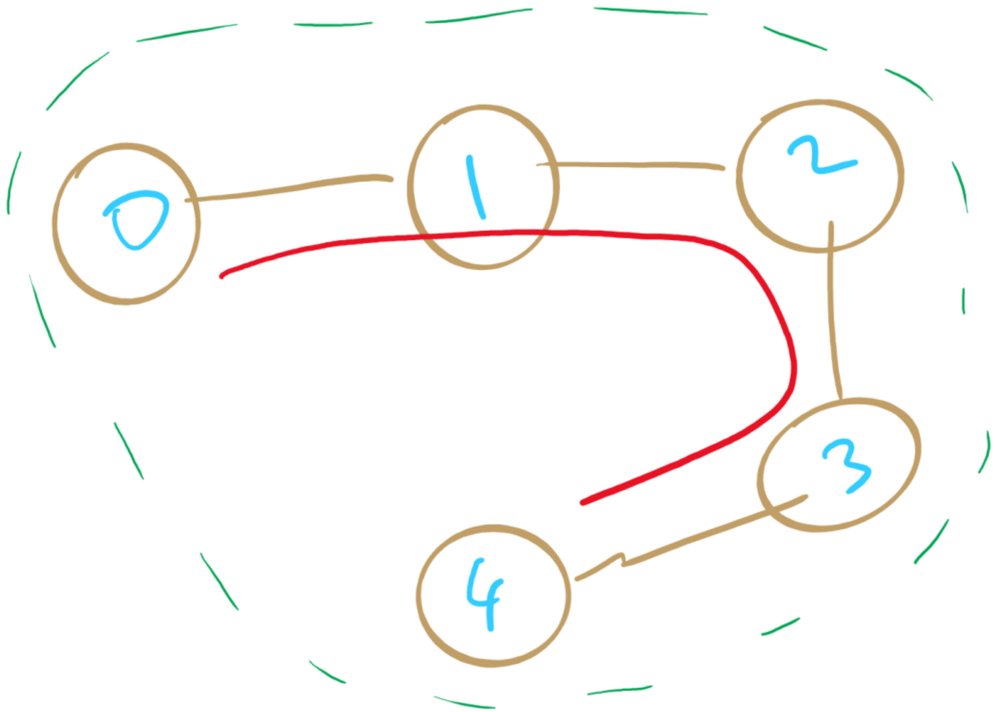


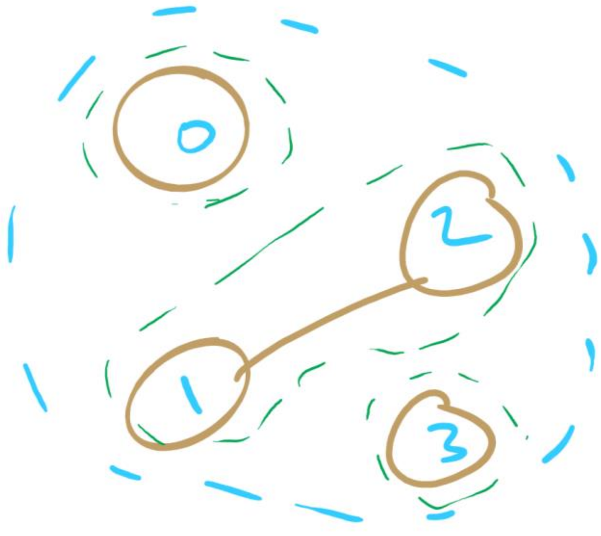
More definitions

- A graph is considered *sparse* if:
  - e <= v lg v
- A graph is considered *dense* as it approaches the maximum number of edges
  - I.e., e == MAX - ε
- A *complete* graph has the maximum number of edges
- Have we seen “sparse” and dense before?



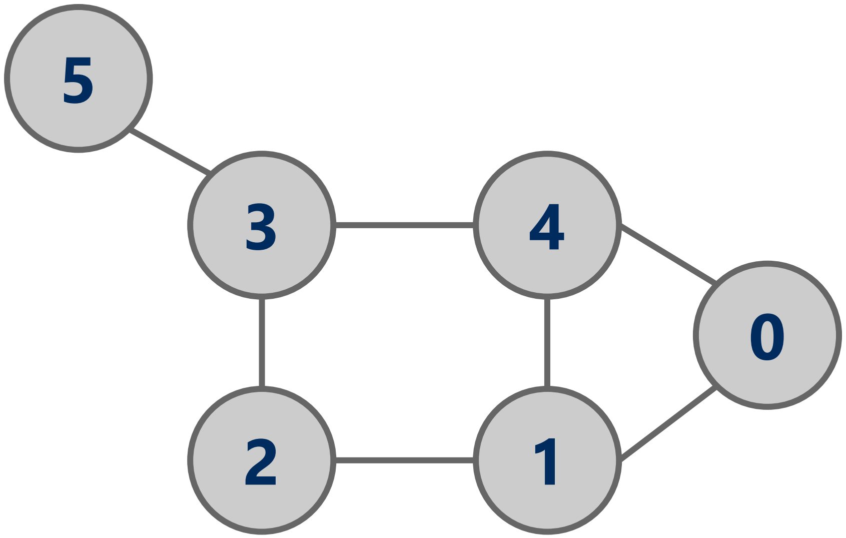
Sparse graphs



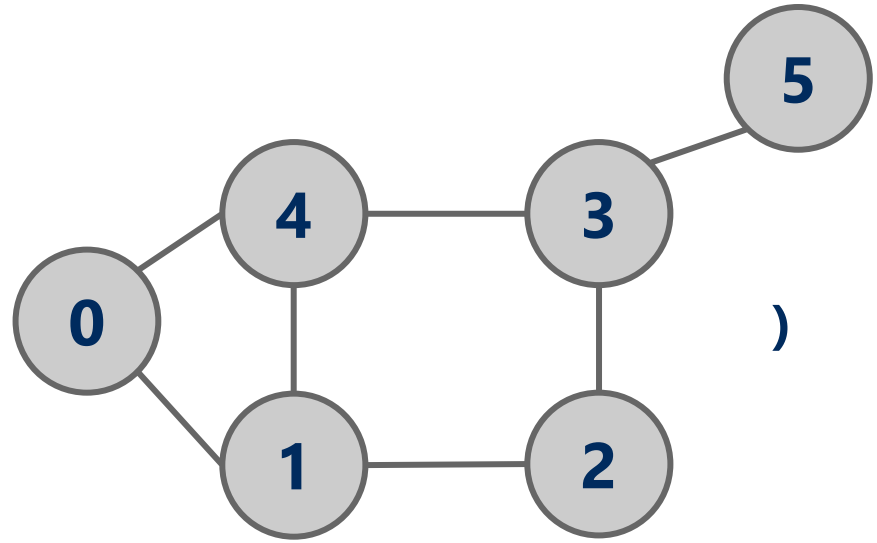


Question:

- Is



.equals(



)

Representing graphs

- Trivially, graphs can be represented as:
  - List of vertices
  - List of edges
- Performance?
  - Assume we're going to be analyzing static graphs
    - I.e., no insert and remove
  - So what operations should we consider?

Graph operations

- Static graphs
  - check if two vertices are neighbors
  - find the list of neighbors of a given vertex
    - for directed graphs, in-neighbors and out-neighbors
- Dynamic graphs
  - add/remove edges
  - Not our focus in this class

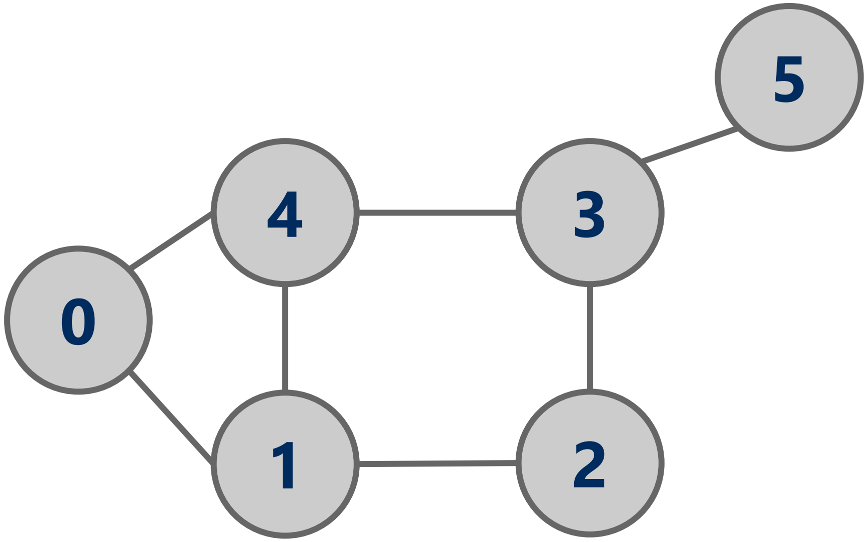
Representing graphs

- Trivially, graphs can be represented as:
    - List of vertices
    - List of edges
- Performance?
    - Check if two vertices are neighbors
        - $O(e)$
    - Find the list of neighbors of a given vertex
        - $O(e)$
- Space?
    - $\Theta(v + e)$ memory

Using an adjacency matrix

- Rows/columns are vertex labels
  - M[i][j] = 1 if (i, j) ∈ E
  - M[i][j] = 0 if (i, j) ∉ E

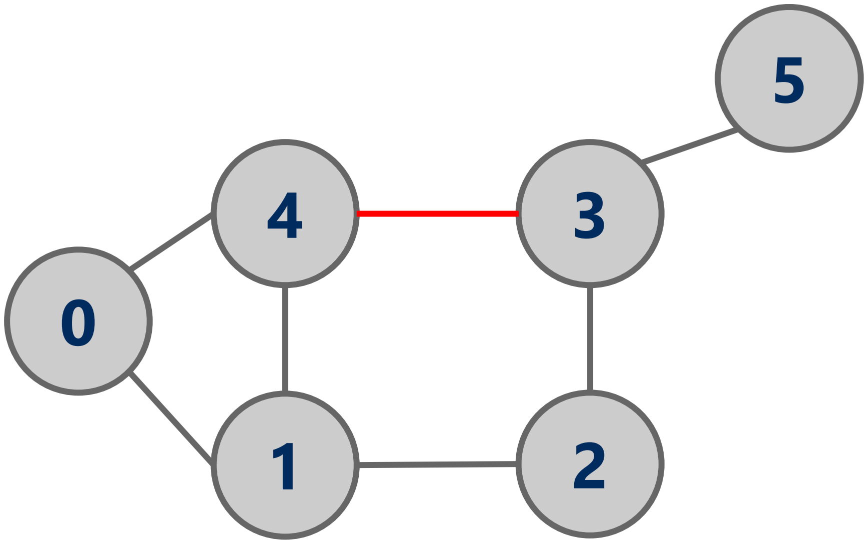
	0	1	2	3	4	5
0	0	1	0	0	1	0
1	1	0	1	0	1	0
2	0	1	0	1	0	0
3	0	0	1	0	1	1
4	1	1	0	1	0	0
5	0	0	0	1	0	0



Using an adjacency matrix

- Rows/columns are vertex labels
  - M[i][j] = 1 if $(i, j) \in E$
  - M[i][j] = 0 if $(i, j) \notin E$

	0	1	2	3	4	5
0	0	1	0	0	1	0
1	1	0	1	0	1	0
2	0	1	0	1	0	0
3	0	0	1	0	1	1
4	1	1	0	1	0	0
5	0	0	0	1	0	0



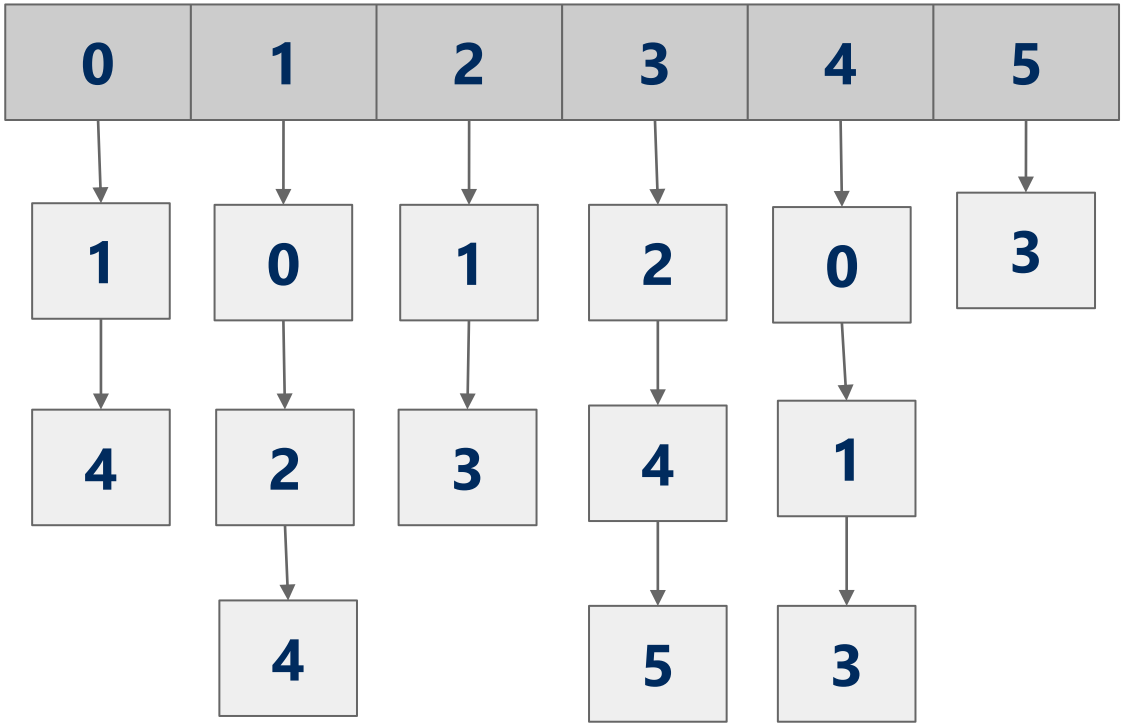
Adjacency matrix analysis

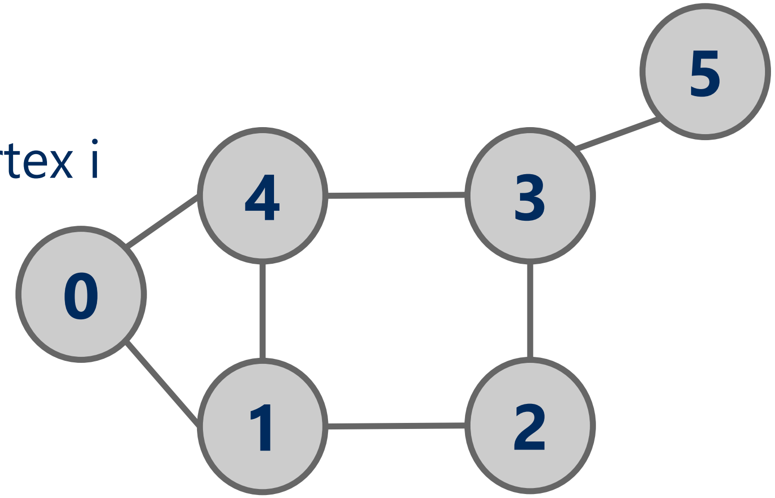
- Runtime?
  - Check if two vertices are neighbors
    - $\Theta(1)$
  - Find the list of neighbors of a vertex
    - $O(v)$
  - $O(v^2)$ time to initialize
- Space?
  - $O(v^2)$

	0	1	2	3	4	5
0	0	1	0	0	1	0
1	1	0	1	0	1	0
2	0	1	0	1	0	0
3	0	0	1	0	1	1
4	1	1	0	1	0	0
5	0	0	0	1	0	0

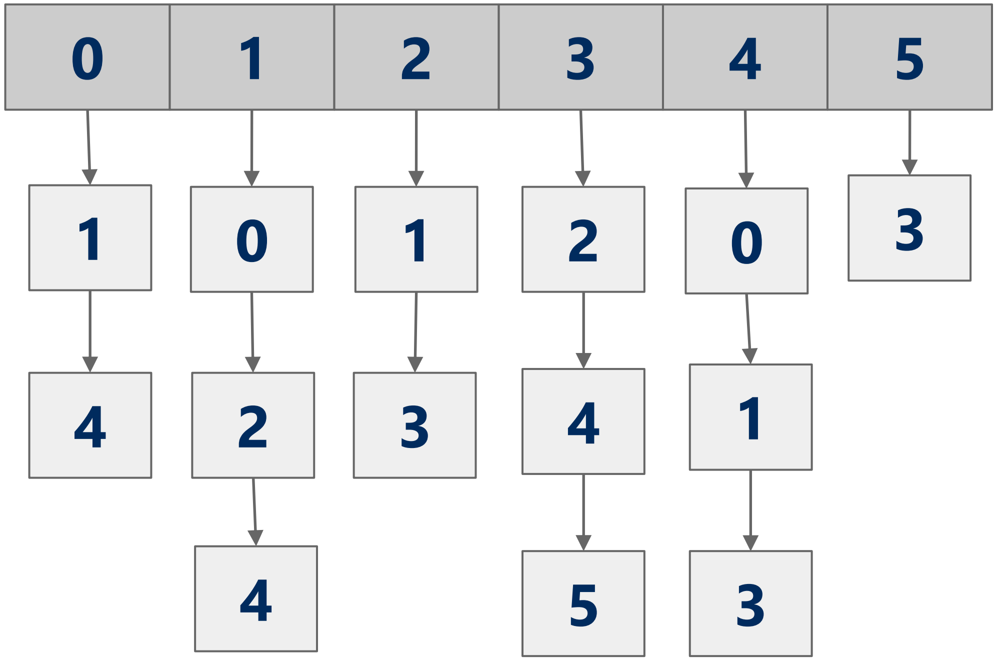
Adjacency lists

- Array of neighbor lists
  - A[i] contains a list of the neighbors of vertex i





Adjacency list analysis



- Runtime?
  - Check if two vertices are neighbors
  - Find the list of neighbors of a vertex
    - Θ(d)
    - d is the degree of a vertex (# of neighbors)
    - O(v)
- Space?
  - Θ(v + e) memory
  - overhead of node use
  - Could be much less than v^2

Comparison

- **Where would we want to use adjacency lists vs adjacency matrices?**
- Dense graphs?
- Sparse graphs?
- **What about the list of vertices/list of edges approach?**

Even more definitions

- Path
  - A sequence of adjacent vertices
- Simple Path
  - A path in which no vertices are repeated
- Simple Cycle
  - A simple path with the same first and last vertex
- Connected Graph
  - A graph in which a path exists between all vertex pairs
- Connected Component
  - Connected subgraph of a graph
- Acyclic Graph
  - A graph with no cycles
- Tree
  - ?
  - A connected, acyclic graph
    - Has exactly v-1 edges

Complete Graph vs. Connected Graph

- Difference between Connected graph and Complete graph?
    - Connected means there is a **path** from A to B for each pair of vertices A and B
    - Complete means there is an **edge** between A and B for each pair of vertices A and B

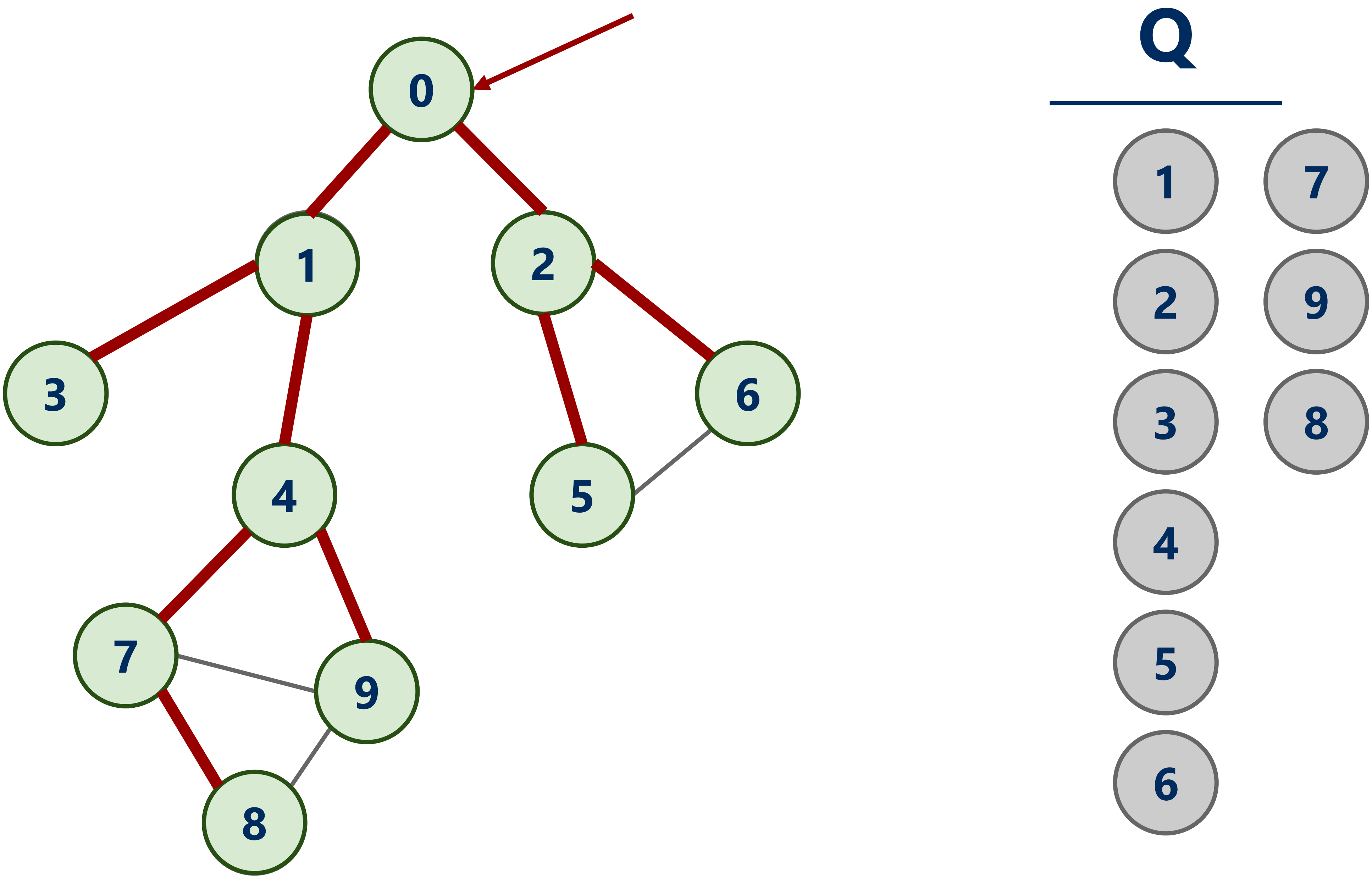
Graph traversal

- What is the best order to traverse a graph?
- Two primary approaches:
  - Breadth-first search (BFS)
    - Search all directions evenly
      - i.e., from i, visit all of i's neighbors, then all of their neighbors, etc.
    - Would help us compute the distance between two vertices
      - Remember our Problem of the Day?
  - Depth-first search (DFS)
    - "Dive" as deep as possible into the graph first
    - Branch when necessary

BFS

- Can be easily implemented using a queue
  - For each vertex visited, add all of its neighbors to the Q (if not previously added)
    - Vertices that have been seen (i.e., added to the Q) but not yet visited are said to be the *fringe*
  - Pop head of the queue to be the next visited vertex
- See example

BFS example



BFS Pseudo-code

```
Q = new Queue

BFS(vertex v){
    add v to Q
    while(Q is not empty){
        w = remove head of Q
        visited[w] = true //mark w as visited
        for each unseen neighbor x
            seen[x] = true //mark x as seen
            parent[x] = w
            add x to Q
    }
}
```

Shortest paths

- BFS traversals can further be used to determine the *shortest path* between two vertices

BFS Pseudo-code to compute shortest paths

```
Q = new Queue

BFS(vertex v){
    add v to Q
    while(Q is not empty){
        w = remove head of Q
        visited[w] = true //mark w as visited
        for each unseen neighbor x
            seen[x] = true //mark x as seen
            parent[x] = w
            distance[x] = distance[w] + 1
            add x to Q
    }
}
```

Problem of the Day

- **Input**: A file containing LinkedIn Connection information formatted like the following:
  - Account1: Connection1, Connection2, …
  - Account2: Connection1, Connection2, …
  - …
- **Output**: Answer the following questions:
  - Given two LI accounts, how "far" are they from each other?
    - E.g., 1^{st} connection, 2^{nd} connection, etc.
  - Are the accounts in the file all ***connected***?
    - If not, how many ***connected components*** are there?
  - Are there certain accounts that if removed, the remaining accounts become ***partitioned***?
    - These account are called ***articulation points***

Problem of the Day

- **Input**: A file containing LinkedIn Connection information formatted like the following:
  - Account1: Connection1, Connection2, …
  - Account2: Connection1, Connection2, …
  - …
- **Output**: Answer the following questions:
  - Given two LI accounts, how "far" are they from each other?
    - E.g., 1st connection, 2nd connection, etc.
  - Are the accounts in the file all ***connected***?
    - If not, how many ***connected components*** are there?
  - Are there certain accounts that if removed, the remaining accounts become ***partitioned***?
    - These account are called ***articulation points***

Finding connected components

- A connected component is a connected subgraph G′
  - (V′, E′)
    - V′ ⊆ V
    - E′ = {(u, v) ∈ E and both u and v ∈ V′}
- To find all connected components:
  - wrapper function around BFS
  - A loop in the wrapper function will have to continually call bfs() while **there are still unseen vertices**
  - Each call will yield a spanning tree for a **connected component** of the graph

BFS Pseudo-code to compute connected components

```
int components = 0
for each vertex v in V
        if visited[v] = false
                components++
                Q = new Queue
                BFS(v)
```

```
BFS(vertex v){
        add v to Q
        component
        while(Q is not empty){
                w = remove head of Q
                visited[w] = true
                component[w] = components
                for each unseen neighbor x
                        seen[x] = true
                        add x to Q
        }
}
```

Problem of previous lecture

- **Input**: A file containing LinkedIn (LI) accounts and their connections
  - Account1: Connection1, Connection2, …
  - Account2: Connection1, Connection2, …
  - …

- **Output**: Answer the following questions:
  - Given two LI accounts, how “far” are they from each other?
    - e.g., 1st connection?, 2nd connection?, etc.
  - Are the accounts in the file all ***connected***?
    - If not, how many ***connected components*** are there?
  - For each connected component, are there certain accounts that if removed, the remaining accounts become ***partitioned***?

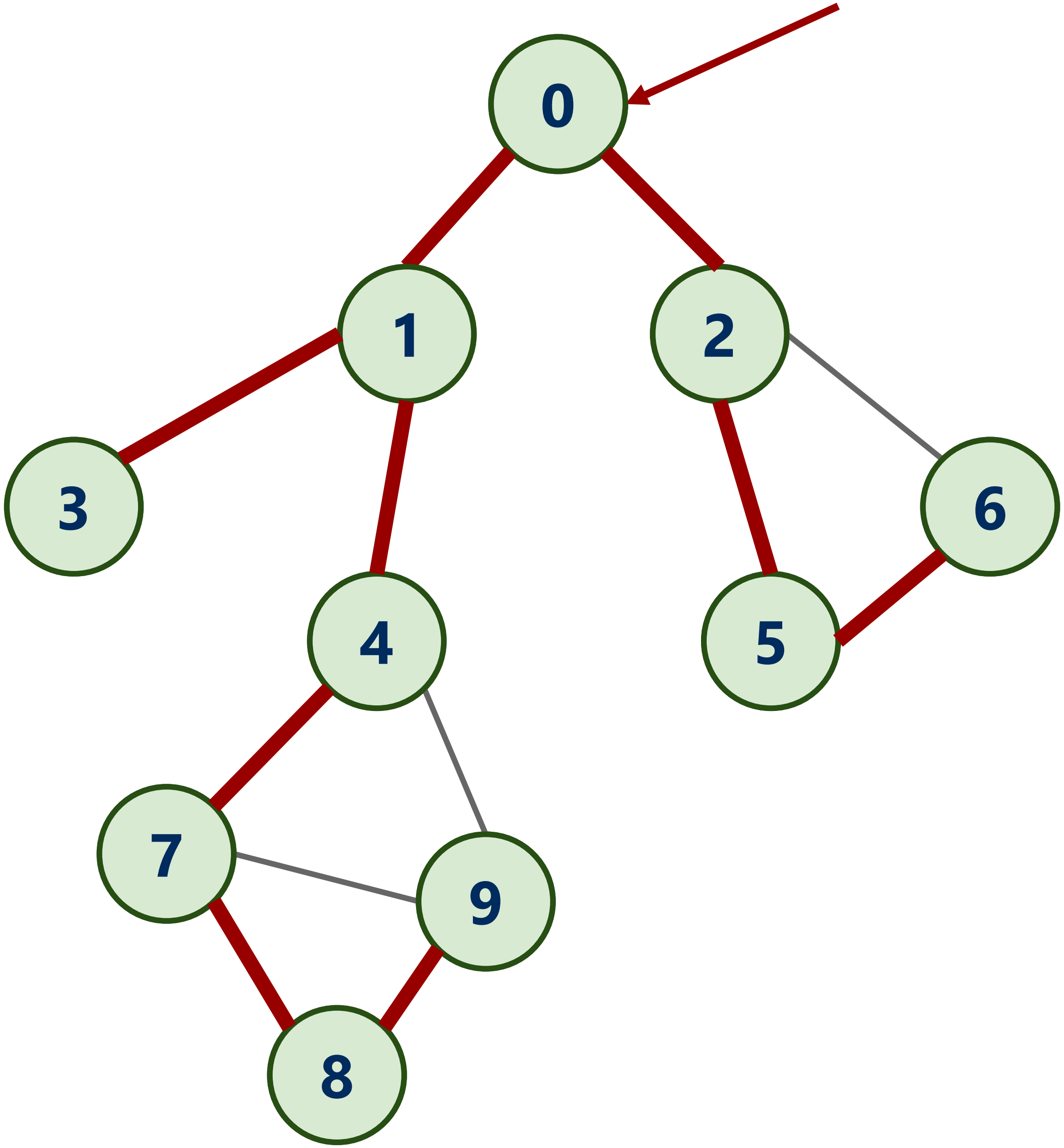
DFS – Depth First Search

- Already seen and used this throughout the term
  - For Huffman encoding…
    - as we build the codebook out of the Huffman Trie
- Can be easily implemented recursively
  - For each vertex, visit *first* unseen neighbor
  - Backtrack at deadends (i.e., vertices with no unseen neighbors)
    - Try *next* unseen neighbor after backtracking
  - An arbitrary order of neighbors is assumed

DFS Pseudo-code

```
DFS(vertex v) {
  seen[v] = true //mark v as seen
  for each unseen neighbor w
    parent[w] = v
    DFS(w)
}
```

DFS example



9
8
6
5
2
0

Runtime Stack

When to visit a vertex

```
DFS(vertex v) {
   seen[v] = true //mark v as seen
   visit v //pre-order DFS
   for each unseen neighbor w
      parent[w] = v
      DFS(w)
}
```

When to visit a vertex

```
DFS(vertex v) {
   seen[v] = true //mark v as seen
for each unseen neighbor w
      parent[w] = v
      DFS(w)
 visit v //post-order DFS
}
```

When to visit a vertex

```
DFS(vertex v) {
   seen[v] = true //mark v as seen
for each unseen neighbor w
      parent[w] = v
      DFS(w)
      (re)visit v //in-order DFS
}
```

Runtime Analysis of BFS

- Each vertex is added to the queue exactly once and removed exactly once
  - v add/remove operations
    - $O(v)$ time for vertex processing
- Edges are checked when adding the list of neighbors to the queue
- Each edge is checked at most twice, one per edge endpoint
  - $O(e)$ time for edge processing
- Total time: vertex processing time + edge processing time
  - $O(v + e)$

Runtime Analysis for DFS

- For Adjacency Matrix representation, BFS checks each *possible* edge!
  - $O(v^2)$ time for edge processing with Adjacency Matrix
- Total time: $O(v^2 + v) = O(v^2)$

Runtime Analysis of DFS

- Each vertex is seen then visited exactly once
  - $O(v)$ time for vertex processing
  - except when (re)visiting a vertex after each child
    - vertex processing happens inside edge processing in that case
- Edges are checked when finding the list of neighbors
- Each edge is checked at most twice, one per edge endpoint
  - $O(e)$ time for edge processing
- Total time: vertex processing time + edge processing time
  - $O(v + e)$

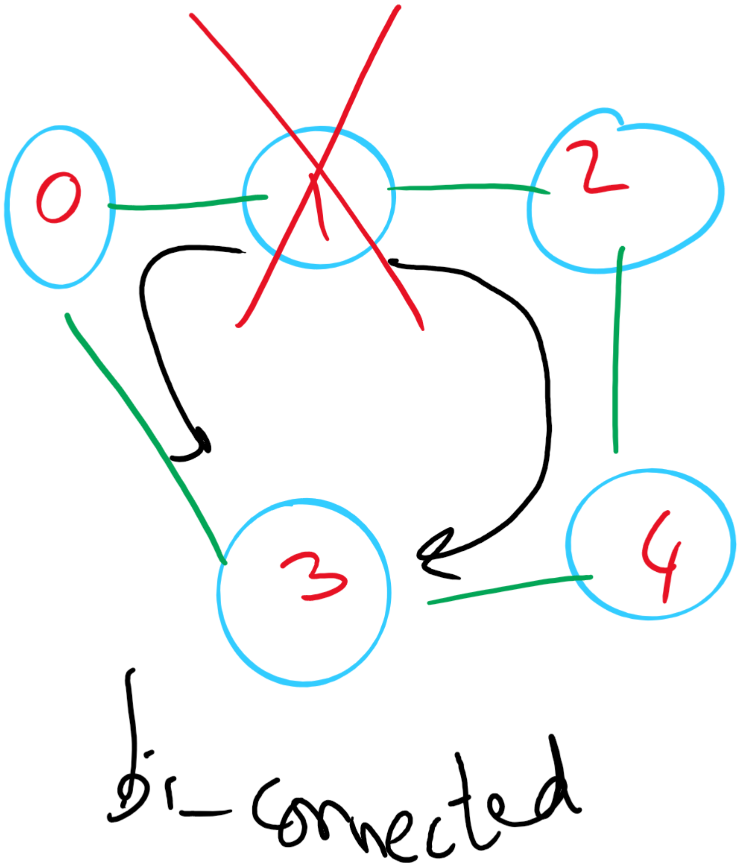
Runtime Analysis of BFS and DFS

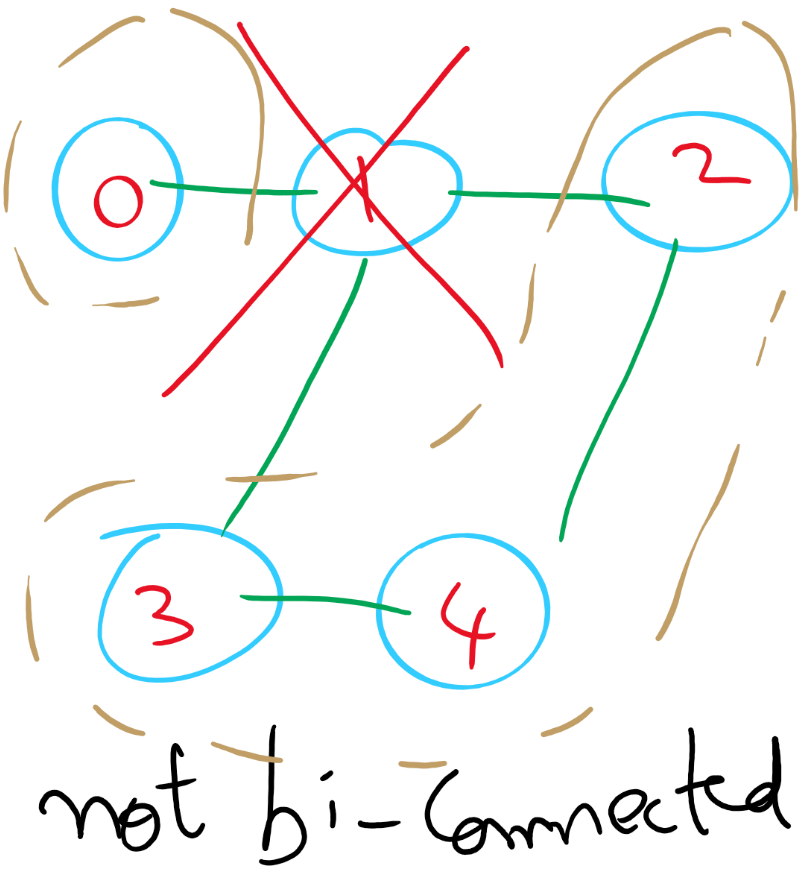
- At a high level, DFS and BFS have the same runtime
  - Each vertex must be seen and then visited, but the order will differ between these two approaches
- The representation of the graph affect the runtimes of of these traversal algorithms?
  - $O(v + e)$ with Adjacency Lists
  - $O(v^2)$ with Adjacency Matrix
  - Note that for a dense graph, $v + e = O(v^2)$

Biconnected graphs

- A *biconnected graph* has at least 2 distinct paths between all vertex pairs
  - a distinct path shares no common edges or vertices with another path except for the start and end vertices
- A graph is biconnected graph iff it has zero *articulation points*
  - Vertices, that, if removed, will separate the graph

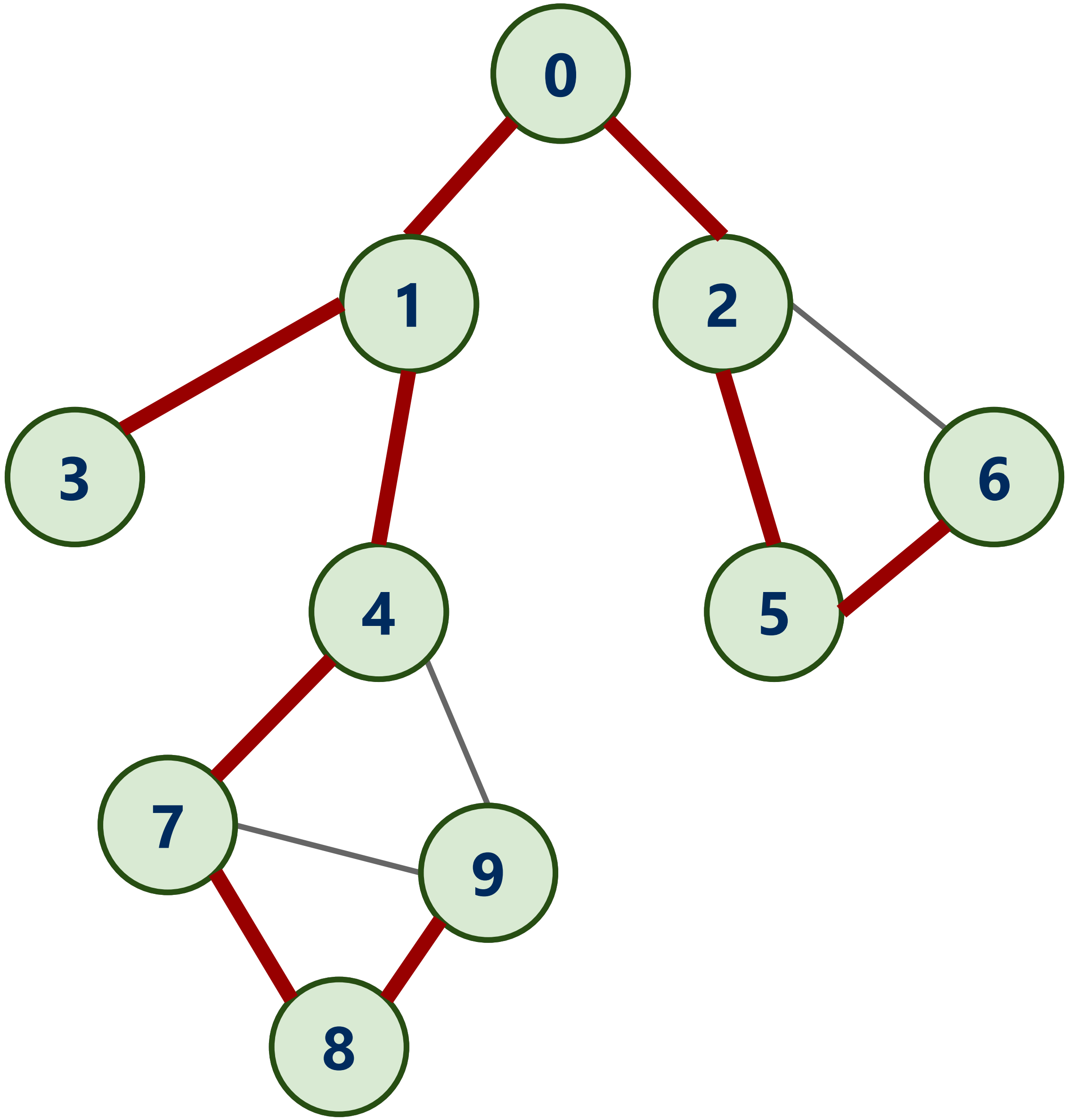
Biconnected Graph





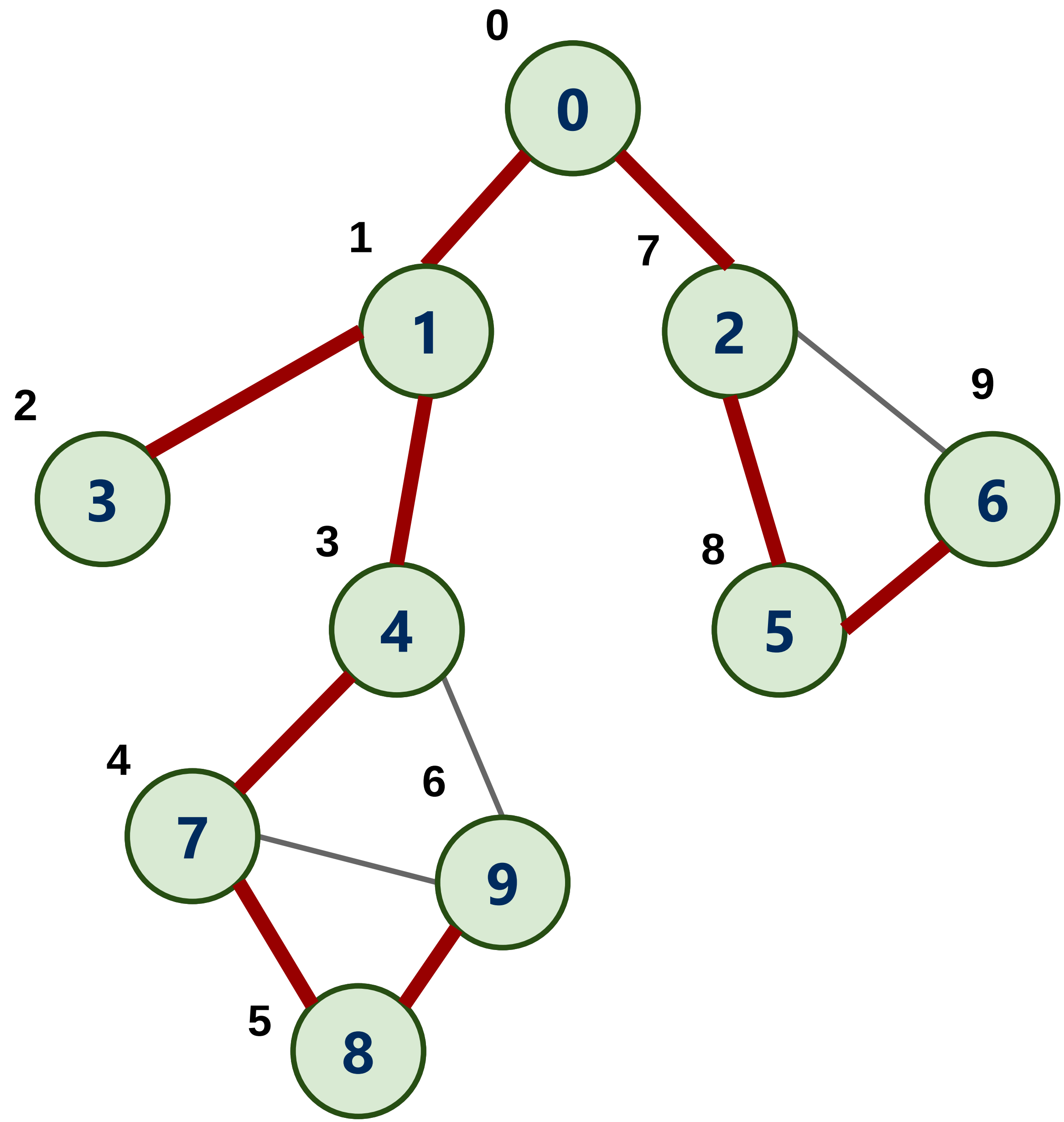
Finding articulation points of a graph

- A DFS traversal builds a spanning tree
  - red edges in the picture
- Edges not included in the spanning tree are called **back edges**
  - e.g., (4, 9) and (2, 6)



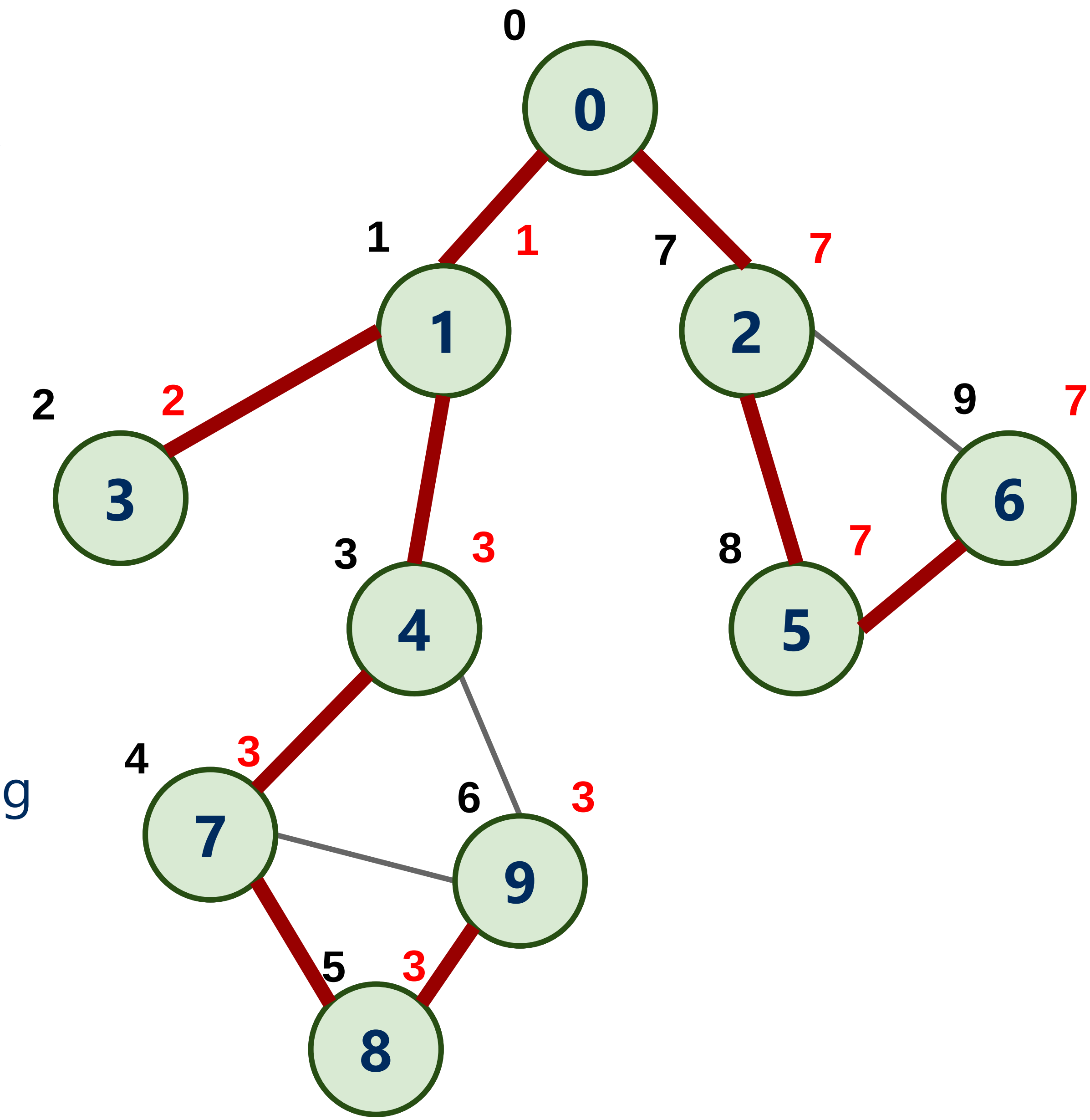
num(v)

- A pre-order DFS traversal visits the vertices in some order
  - let's number the vertices with their traversal order
  - num(v)



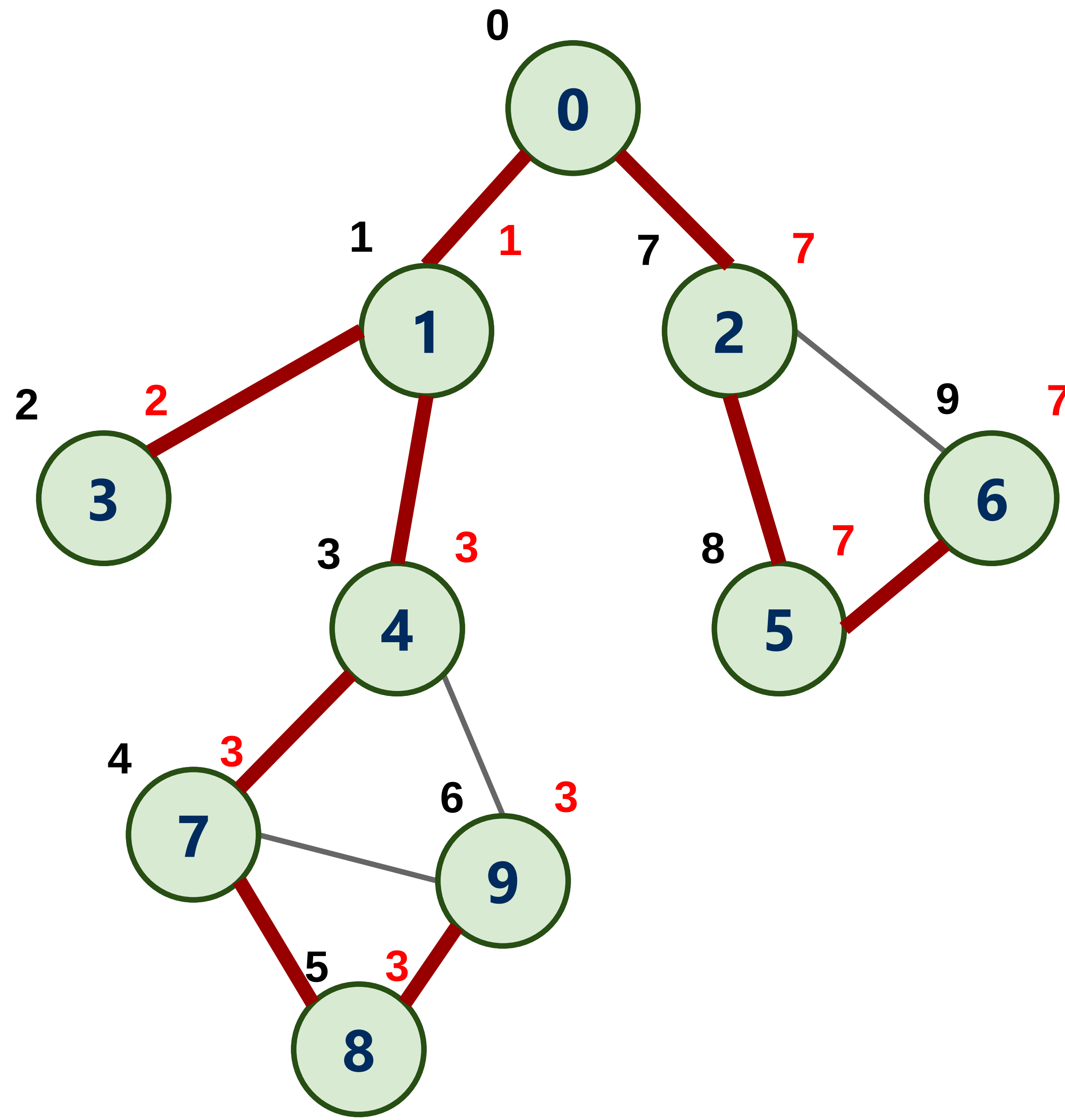
Finding articulation points of a graph

- For each non-root vertex v, find the lowest numbered vertex reachable from v
  - **not through v's parent**
  - **using 0 or more tree edges then at most one back edge**
- move down the tree looking for a back edge that goes backwards the furtheset



low(v)

- How do we find low(v)?
- low(v) = Min of:
  - num(v)
  - num(w) for all back edges (v, w)
  - low(w) of all children of *v*

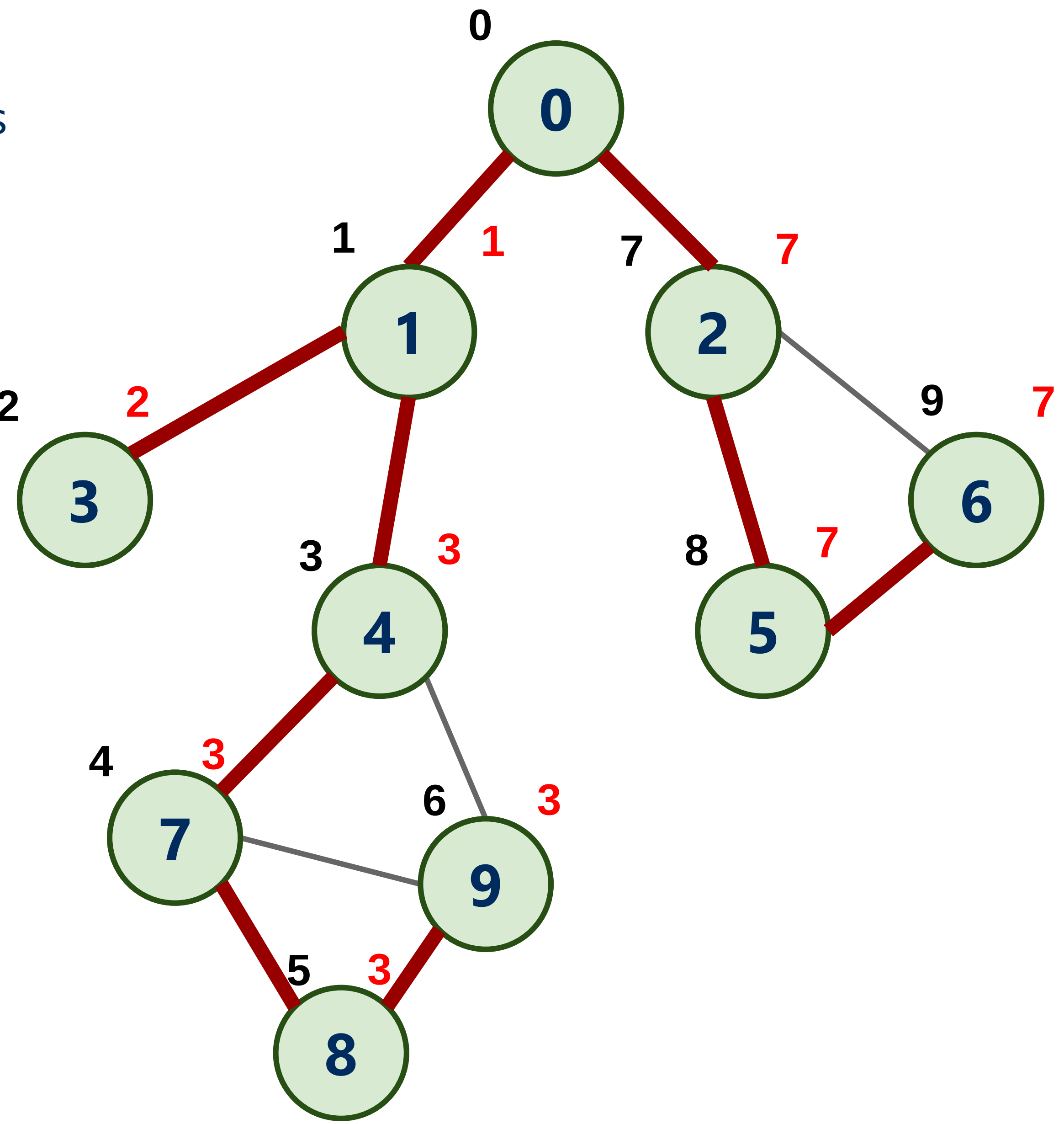


low(v)

- low(v) = lowest-numbered vertex reachable from v using 0 or more spanning tree edges and then **at most one** back edge
  - Min of:
    - num(v) (the vertex is reachable from itself)
    - Lowest num(w) of all back edges (v, w)
    - Lowest low(w) of all children of *v* (the lowest-numbered vertex reachable through a child)

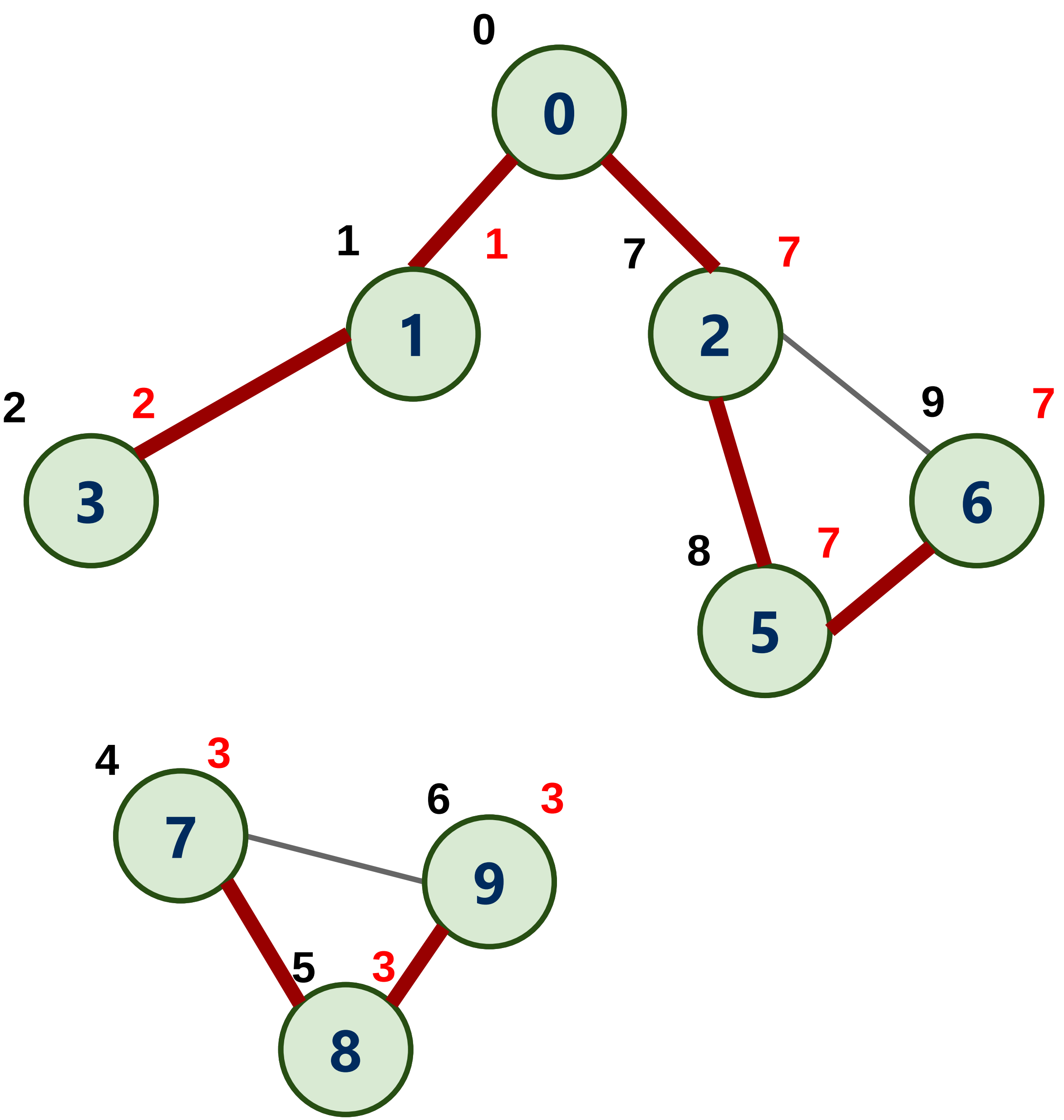
Why are we computing low(v)?

- What does it mean if a vertex has a child such that
  - **low(child) >= num(parent)**?
- e.g., 4 and 7
- child has **no other way** except through parent to reach vertices with lower num values than parent
- e.g., 7 cannot reach 0, 1, and 3 except through 4
- So, the **parent is an articulation point**!
  - e.g., if 4 is removed, the graph becomes disconnected



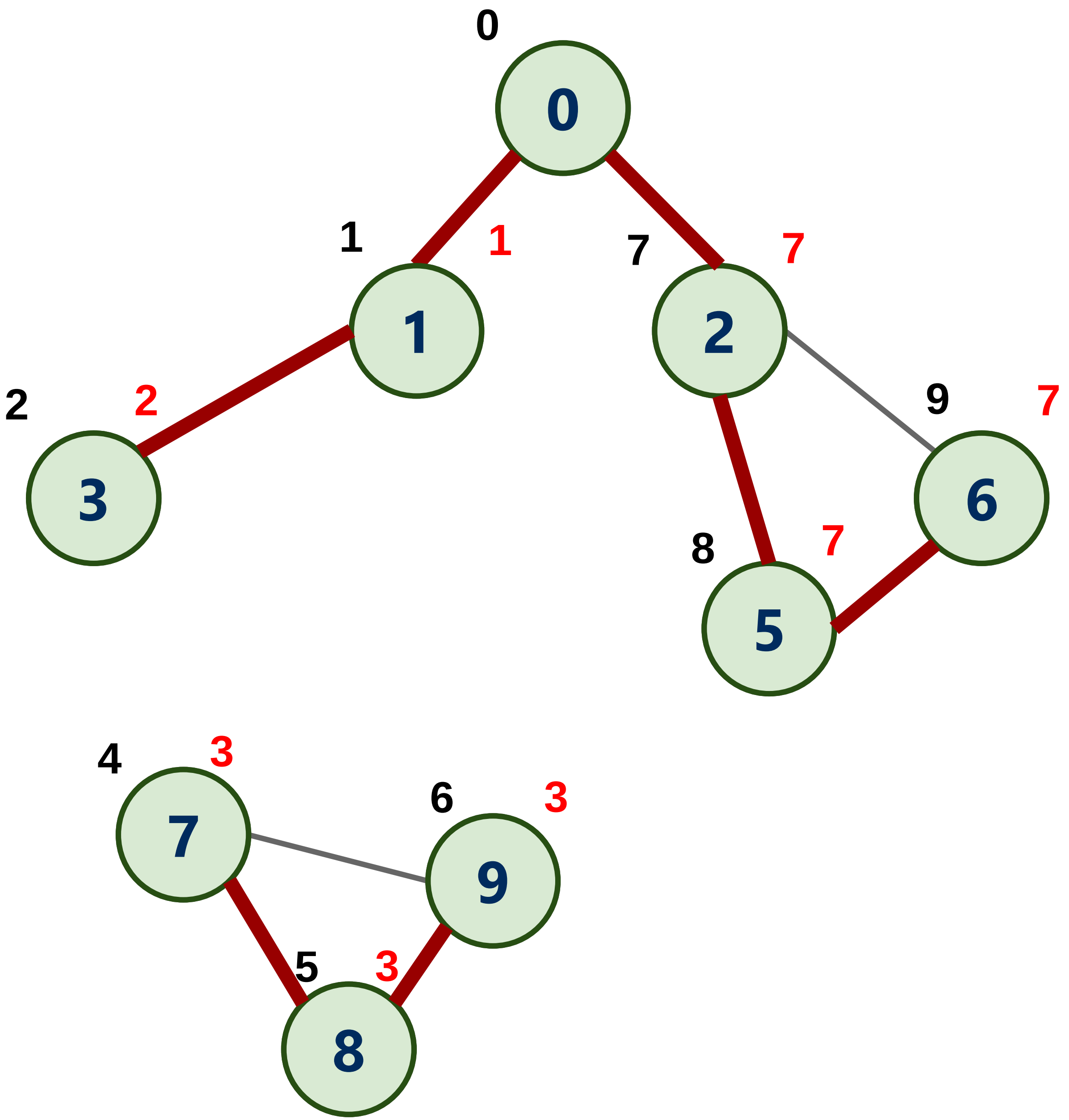
Why are we computing low(v)?

- if 4 is removed, the graph becomes disconnected
- Each **non-root vertex v** that has a child w such that **low(w) >= num(v) is an articulation point**



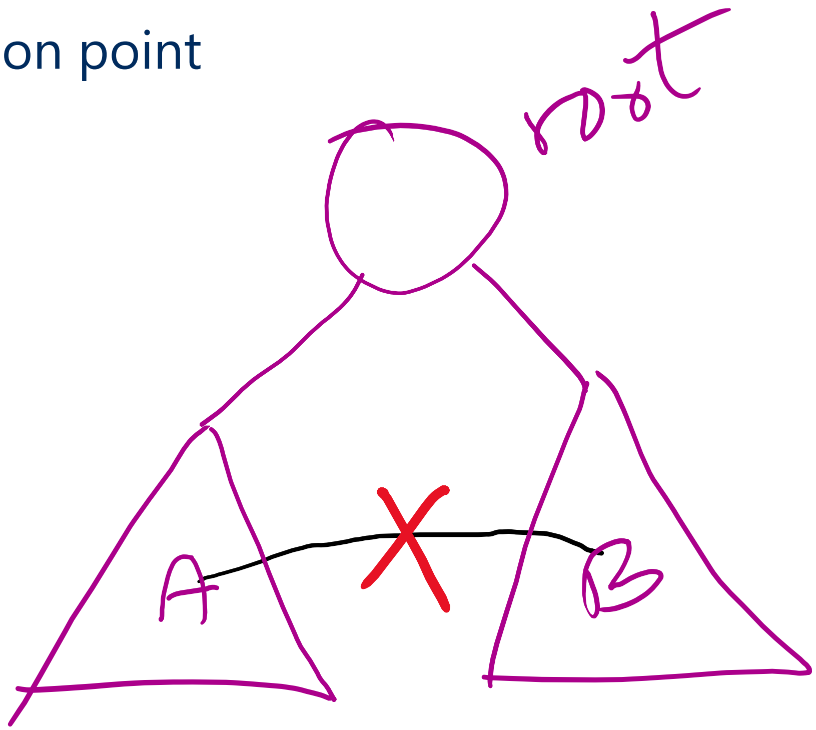
What about the root vertex?

- The root has the smallest num value
  - root's children can't go "further" than root
- Possible that low(child) == num(root) but root is not an articulation point
- need a different condition for root



What about the root of the spanning tree?

- What if we start DFS at an articulation point?
  - The starting vertex becomes the root of the spanning tree
  - If the root of the spanning tree has more than one child, the root is an articulation point



Finding articulation points of a graph: The Algorithm

- As DFS visits each vertex v
  - Label v with with the two numbers:
    - num(v)
    - low(v): initial value is num(v)
  - For each neighbor w
    - if already seen → we have a back edge
      - update low(v) to num(w) if num(w) is less
    - if not seen → we have a child
      - call DFS on the child
      - **after the call returns,**
        - update low(v) to low(w) if low(w) is less

when to compute num(v) and low(v)

- num(v) is computed as we move down the tree
  - pre-order DFS
- low(v) is updated as we move down and up the tree
- Recursive DFS is convenient to compute both
  - why?

Using DFS to find the articulation points of a connected undirected graph

```
int num = 0
DFS(vertex v) {
    num[v] = num++
    low[v] = num[v] //initially
    seen[v] = true //mark v as seen
    for each neighbor w
       if(w unseen){
          parent[w] = v
          DFS(w) //after the call returns low[w] is computed, why?
          low[v] = min(low[v], low[w])
          if(low[w] >= num[v]) v is an articulation point
       } else { //seen neighbor
          if(w != parent[v]) //and not the parent, so back edge
            low[v] = min(low[v], num[w])
       }
```

Finding articulation points example